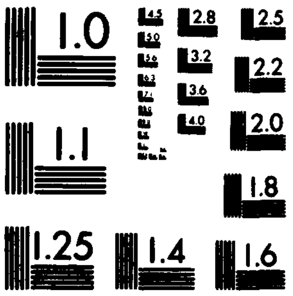

1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3
7.1
8.0

RADC-TR-81-358, Vol IV (of seven)
Interim Report
December 1981

ADA INTEGRATED ENVIRONMENT I COMPUTER PROGRAM DEVELOPMENT SPECIFICATION

Intermetrics, Inc.

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, New York 13441

This document was produced under Contract F30602-80-C-0291 for the Rome Air Development Center. Mr. Don Roberts is the COTR for the Air Force. Dr. Fred H. Martin is Project Manager for Intermetrics.

This report has been reviewed by the RADC Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-81-358, Volume IV (of seven) has been reviewed and is approved for publication.

APPROVED: [illegible]
DONALD F. ROBERTS
Project Engineer

APPROVED: [illegible]
JOHN J. MARCINIAK, Colonel, USAF
Chief, Command and Control Division

FOR THE COMMANDER: John P. Huss
JOHN P. HUSS
Acting Chief, Plans Office

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER RADC-TR-81-358, Vol IV (of seven)	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) ADA INTEGRATED ENVIRONMENT I COMPUTER PROGRAM DEVELOPMENT SPECIFICATION		5. TYPE OF REPORT & PERIOD COVERED Interim Report 15 Sep 80 - 15 Mar 81
		6. PERFORMING ORG. REPORT NUMBER N/A
7. AUTHOR(s)		8. CONTRACT OR GRANT NUMBER(s) F30602-80-C-0291
9. PERFORMING ORGANIZATION NAME AND ADDRESS Intermetrics, Inc. 733 Concord Avenue Cambridge MA 02138		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 62204F/33126F 55811908
11. CONTROLLING OFFICE NAME AND ADDRESS Rome Air Development Center (COES) Griffiss AFB NY 13441		12. REPORT DATE December 1981
		13. NUMBER OF PAGES 60
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Same		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Same

18. SUPPLEMENTARY NOTES

RADC Project Engineer: Donald F. Roberts (COES)

Subcontractor is Massachusetts Computer Assoc.

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Ada	MAPSE	AIE
Compiler	Kernel	Integrated environment
Database	Debugger	Editor
KAPSE	APSE	

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The Ada Integrated Environment (AIE) consists of a set of software tools intended to support design, development and maintenance of embedded computer software. A significant portion of an AIE includes software systems and tools residing and executing on a host computer (or set of computers). This set is known as an Ada Programming Support Environment (APSE). This B-5 Specification describes, in detail, the design for a minimal APSE, called a MAPSE. The MAPSE is the foundation upon which an

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

APSE is built and will provide comprehensive support throughout the design, development and maintenance of Ada software. The MAPSE tools described in this specification include an Ada compiler, linker/loader, debugger, editor, and configuration management tools. The kernel (KAPSE) will provide the interfaces (user, host, tool), database support, and facilities for executing Ada programs (runtime support system).

TABLE OF CONTENTS

1.0 SCOPE

1.1 Identification

This specification defines the requirements for MAPSE Generation and Support (MGS). The functional area identified by the term MGS consists of those elements used in the construction, maintenance, and rehosting of MAPSE tools (as opposed to programs that are primary MAPSE tools).

1.2 Functional Summary

MGS is the means by which MAPSE tools are constructed and maintained in a consistent, reliable, and portable form. Within this functional area, three sub-areas are defined: (1) Text Parsing; (2) Data Management; and (3) Initial Tool Construction.

1.2.1 Text Parsing (Lexer/Parser Generator)

Tools that must parse text input (notably the compiler) are in part constructed "automatically"; that is, by means of lexer and parser generators. This assures a correct parse and provides uniform treatment of syntactic errors. By making the lexer and parser generators MAPSE tools, the maintenance of existing tools and generation of new tools is supported entirely within the MAPSE.

1.2.2 Data Management (Virtual Memory Methodology)

MAPSE tools must be able to preserve data structures within the KAPSE Database in order to communicate with other tools or with subsequent activations of the same tool. In general, it cannot be assumed that the address space of the host machine will be adequate to keep such data structures entirely within memory while they are used. A Virtual Memory Methodology (VMM) provides both a means of representing the data structures used by tools in a consistent and efficiently-accessed external form, and a means of overcoming address space limitations on the size of data structures. In addition, VMM provides aids to debugging (a human readable representation) and communication between hosts (a compressed binary host-independent representation).

1.2.3 Initial Tool Construction (Bootstrap)

Since MAPSE tools, including the Ada compiler, are coded in Ada, a bootstrap step is required to establish an operational self-hosted compiler, as it is for any compiler implemented in the language it compiles. However, the runtime environment required to execute the MAPSE compiler is supplied by MAPSE and KAPSE functions which are themselves coded in Ada. Establishing a complete operational MAPSE requires a number of intermediate bootstrapping steps to support parallel development of the compiler, the KAPSE, and the separate compilation facilities of the MAPSE.

The steps in such a parallel development include rehosting the compiler and other tools from an intermediate development environment to run in the MAPSE environment; these are essentially the same steps involved in rehosting the completed MAPSE to a different machine (excluding code generator modifications in the host compiler). Therefore the subject of initial tool construction is covered in detail in Section 3.3 where rehosting considerations are included.

2.0 APPLICABLE DOCUMENTS

Please note that the bracketed number preceding the document identification is used for reference purposes within the text of this document.

2.1 Government Documents

[G-1] Reference Manual for the Ada Programming Language, proposed standard document, U.S. Department of Defense, July 1980.

2.2 Non-Government Documents

[N-1] Diana Reference Manual, G. Goos and Wm. Wulf, Institut Fuer Automatik II, Universitaet Karlsruhe and Computer Science Dept, Carnegie-Mellon University, March 1981.

[N-2] Guido Persch, Georg Winterstein, Manfred Dausmann, Sophia Drossopoulu, AIDA Implementation Description, Institut fuer Automatik II, Universitaet Karlsruhe, December 1980.

[N-3] Wetherell, Charles, Alfred Shannon, LR: Automatic Parser Generator and LR(1) Parser, Preprint UCRI-82926, University of California, Davis, June 1979.

[N-4] Pager, D., A Practical General Method for Constructing LR(k) Parsers, Acta Informatica 7, pp.249-268, 1977.

[N-5] Manfred Daussmann, Sophia Drossopoulu, Guido Persch, Georg Winterstein, An Informal Introduction to AIDA, Institut fuer Automatik II, Universitaet Karlsruhe, November 1980.

[N-6] Benjamin M. Brosgol, David Alex Lamb, David R. Levine, Joseph M. Newcomer, Mary S. Van Deusen, William A. Wulf, TCOL_{Ada}: Revised Report on an Intermediate Representation for the Preliminary Ada Language, Computer Science Dept, Carnegie-Mellon University, February 1980.

[N-7] R. Cattell, D. Dill, P. Hilfinger, S. Hobbs, B. Leverett, J. Newcomer (principal editor), A. Reiner, B. Schatz, W. Wulf, PQCC Implementor's Handbook, October 1980.

[N-8] J. D. Ichbiah, J. G. P. Barnes, J. C. Helian, B. Krieg-Bruechner, O. Roubine, B.A. Wichmann, Rationale for the Design fo the Ada Programming Language; ACM SIGPLAN Notices, Vol. 14, No.6, June 1979, Part B.

[I-1] Intermetrics LG Description, 31 August, 1980, IR-536

[I-2] LG User's Guide, December 1979, IR-427

[I-3] System Specification for Ada Integrated Environment, Type A, Intermetrics, Inc., March 1981, IR-676.

Computer Program Development Specifications for Ada Integrated Environment (Type 5):

[I-4] Ada Compiler Phases, IR-677

[I-5] KAPSE/Database, IR-678

[I-6] MAPSE Command Processor, IR-679

[I-7] Program Integration Facilities, IR-681

[I-8] MAPSE Debugging Facilities, IR-682

[I-9] MAPSE Text Editor, IR-683

[I-10] Technical Report (Interim) IR-684

3.0 REQUIREMENTS

3.1 Program Definition

3.1.1 Text Parsing (Lexer/Parser Generators)

3.1.1.1 Capabilities

MAPSE tools that parse text input use a table-driven lexer and LR parser. A lexer is employed to break an input file into a stream of tokens. The lexer accepts some language from the family of regular expressions, i.e., it is equivalent to a finite state automaton. The parser takes the stream of tokens produced by the lexer and produces a parse of the input. The LR parser accepts some language from the family of LR (1) languages. These capabilities are required by the compiler, which contains the most heavily used parser in the MAPSE, LEXSYN [I-4,3.2.2].

Two tools are utilized to create the parser and lexer - the parser generator and the lexer generator. Each tool consists of:

1. a program whose input is a formal grammar and whose output is a set of tables; and

2. a skeleton recognizer program that works from these tables.

3.1.1.2 Interfaces

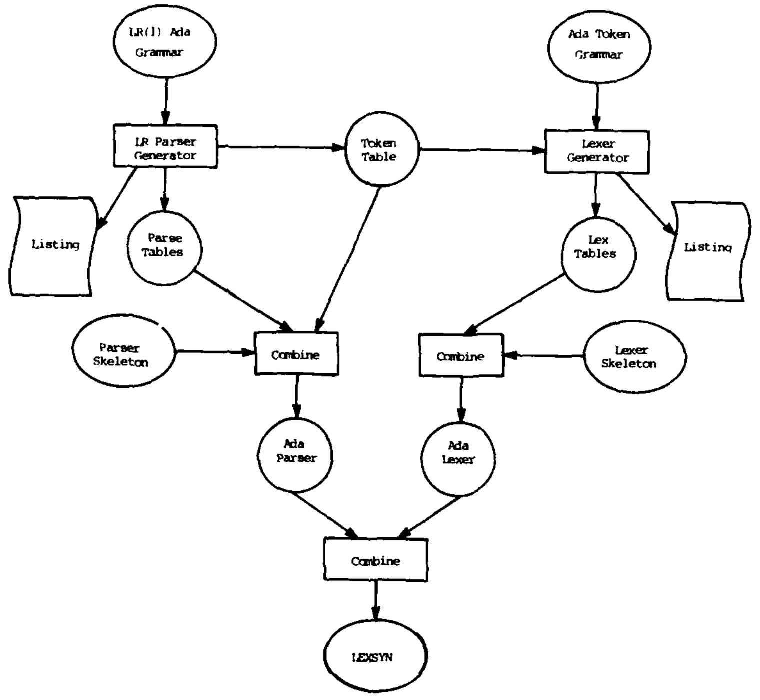
Figure 3-1 displays the offline operation of creating an Ada parser. It demonstrates the interfaces of both generators, especially the token table that is passed from the parser generator to the lexer generator. The specific parameters and methods of processing are described in Section 3.2.1.

3.1.2 Data Management (Virtual Memory Methodology)

3.1.2.1 Capabilities

The VMM is a technique for defining, creating, and accessing representations of data structures. The use of Virtual Memory Methodology to implement a data structure in Ada provides the following capabilities:

1. A permanent, directly accessible representation of an instance of a data structure can be created in the KAPSE database and can be efficiently accessed by any MAPSE tool that uses the same definition of the data structure,



21281131-12

Figure 3-1: Creation of LEXSYN by Lexer and Parser Generators

▭ *indicates processing*

○ indicates data

→ indicates data flow

FIGURE 3-1: CREATION OF LEXSYN BY LEXER AND PARSER GENERATORS

2. Since data paging is part of the access method, the direct addressability of such data structures is independent of the actual addressing range of the host system.

3. It is possible to perform automatic conversions between a directly accessible representation of a data structure and either of two host system independent linear representations:

   a) a human-readable text, which is primarily of use for debugging and testing; or

   b) a compressed binary form, which may be used to transfer a representation between hosts.

While the human-readable form could also be used for the latter purpose, compressed binary is much more compact and uses fewer resources for the conversion process.

3.1.2.2. Interfaces

(a) KAPSE. VMM interfaces with the KAPSE database via package Input_Output. In particular, VMM instantiates the package for a single record type (which is known as a VMM page) and creates dynamic objects that include objects of the type IN_FILE and INOUT_FILE from that instantiation.

The string names for database objects, required by certain Input/Output operations, are obtained in one of the following ways: (1) as actual parameters to VMM operations that invoke Input/Output operations; (2) as values from database objects whose names were previously supplied as actual parameters to VMM operations; or (3) as uniquely-generated names for temporary database objects.

VMM also operates on IN_FILE and OUT_FILE objects of package TEXT_IO. These files are used for error messages, human-readable representations, and compressed binary representations.

(b) Tool Builder. Since VMM is a method for implementing data structures used by MAPSE tools, it defines a "user interface" to the tool builder. This interface has three main components: (1) data structure definitions; (2) primitive operations; and (3) linear representations.

DATA STRUCTURE DEFINITIONS:. Data structure definitions describe the data structures to be accessed by the tool. The definition must provide sufficient information so that the tool can access the data as an Ada object, and so that VMM operations can be derived to create the object and/or convert it to or from a human-readable representation. To accomplish this, the tool builder uses an Ada package specification to define a virtual record type. This virtual record type is specified as an Ada variant record type with a single discriminant. The discriminant must be of an enumeration type defined in the same package.

A virtual record is named by a literal belonging to the enumeration type, and consists of those components that are applicable to the variant record having that value as a discriminant. This is the basis for mapping between internal and human-readable forms.

The types of virtual record components are limited to a subset of Ada type constructors and a set of types supplied by the VMM implementation through generic packages. In particular, no Ada ACCESS types may be used; instead, VMM locator types are used.

While the definition package specification coded by the tool builder is valid Ada, that package specification is not directly compiled with the tool. Instead, the specification is processed by a representation analyzer that combines the definitions for one or more virtual record types, enforces the restrictions and conventions required by VMM, and generates a new package specification and package body. The package is called the VMM access package, and all operations on VMM data that are available to the MAPSE tool are declared in its visible part (see Figure 3-2). The new package specification includes additional declarations that are needed by VMM but which would be tedious to code explicitly. The package body defines procedures that build a symbolic description of the virtual records (i.e., defines character strings for identifiers) to support the reading and writing of the human readable form.

PRIMITIVE OPERATIONS: Primitive operations are defined in the VMM access package produced by the representation analyzer. A MAPSE tool must be compiled with this package in order to create, access, and operate on a VMM data structure. The operations declared in the access package are generally supported by more primitive operations defined in the VMM implmentation package which are, in turn, supported by package input_output in a layered fashion. A MAPSE tool may very well interpose another layer of abstraction on top of the access package as shown in Figure 3-3.

Most primitive operations require a reference to a virtual memory domain (VMD), a collection of virtual memory sub-domains (VMSD) that contain VMM objects described by the same VMM data definition package. Those objects may also contain VMM locators that directly identify VMM objects in any other sub-domain. A VMM object may be created within the context of a VMSD, yielding a VMM locator. Within the context of a VMD, a VMM locator may be dereferenced to obtain an Ada ACCESS value known as a VMM accessor. The type of this value is defined by the representation analyzer as an unconstrained Ada ACCESS type to the variant record type specified in a virtual record type definition. In general, the tool builder uses the dereference and convert-to-ACCESS-value technique to access virtual record components.

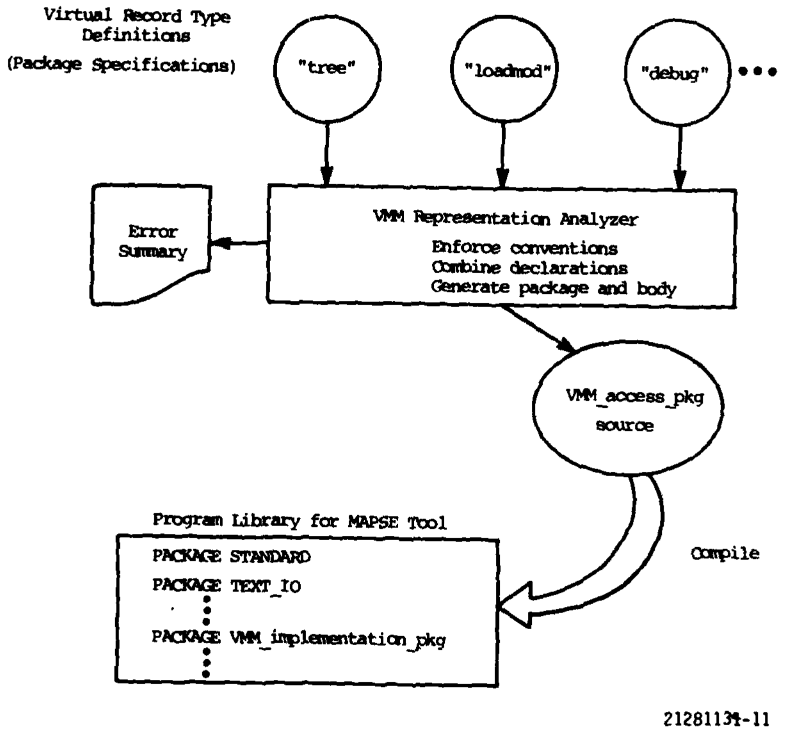


Figure 3-2: VMM Representation Analyzer

FIGURE 3-2: VMM REPRESENTATION ANALYZER

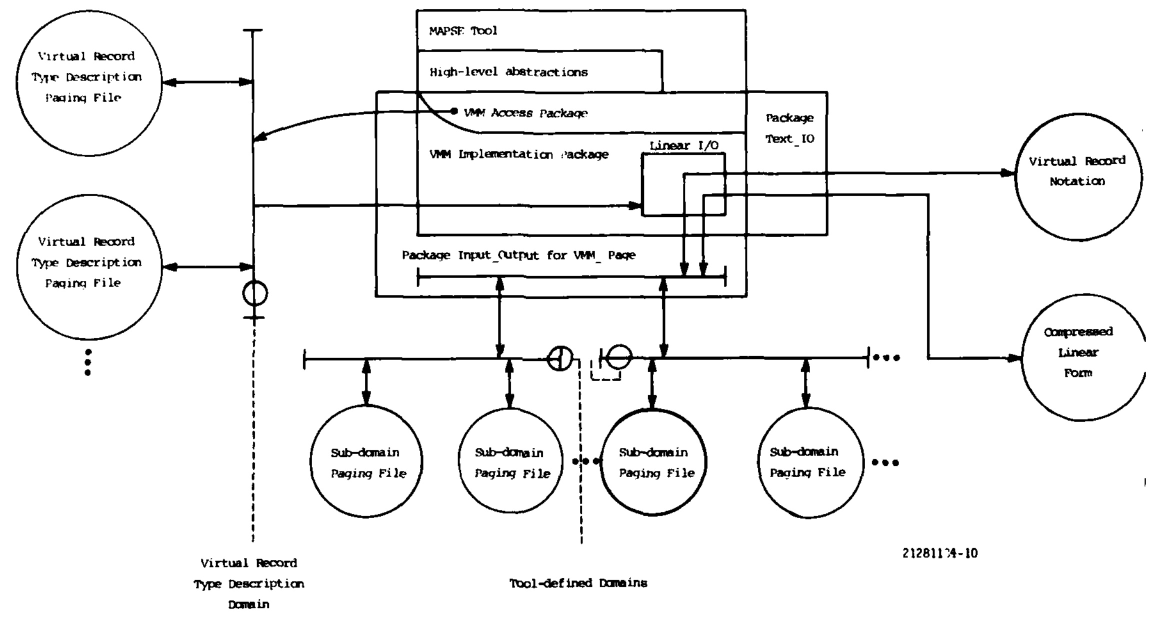


Figure 3-3: VMM Package Hierarchy and Data Paths

FIGURE 3-3: VMM PACKAGE HIERARCHY AND DATA PATHS

While a VMM locator, once obtained, remains valid throughout the life of a VMD, a VMM accessor is valid only within a smaller dynamic scope of program execution. That scope extends from the time at which the accessor was given a value (by dereferencing a VMM locator) to the time at which the next locator is dereferenced (in the context of any VMD anywhere in the program). The validity scope for a VMM accessor may also be explicitly controlled.

Other primitive operations are defined for a number of VMM data abstractions. These include lists, arrays, and sets of virtual records and object types defined by the VMM implementation, as well as varying length strings. In general, these operations use or produce typed VMM locators to designate operands and results; these locators, unlike locators for virtual records, are not dereferenced by the tool builder.

LINEAR REPRESENTATIONS: Linear representations of VMM domains can be generated which represent VMM objects in terms of name => value associations. These linear forms can also be read by VMM programs which use virtual record types with compatible names. This facility is used by the tool builder to display and generate instances of data structures during tool development and testing, and thus presents an interface to the tool builder. Because the human readable form is simply a text file, it can be easily manipulated using the MAPSE text editor.

A compressed binary representation, semantically equivalent to the human-readable form but more compact, is manipulated only by VMM operations and does not present an interface known outside of VMM.

The human-readable representation of VMM data structures is loosely based on the form proposed for the external representation of the Diana intermediate language for Ada programs [References N-1 and I-4]. This form is an adaptation of the linear graph notation used for the TCOL_{Ada} intermediate language [N-6]; the primary textual differences of interest are the use of explicit bracket tokens which permit nested forms, and a textual distinction between defining occurrences and references. The form used to represent VMM domains reflects the sub-domain structure explicitly and contains other constructs found to be useful in the linear graph notation used to represent virtual memory data structures in existing Intermetrics' compilers. The details of the adaptation for use by VMM are provided in Section 3.2.2.5. This form of representation is called virtual record notation (VRN).

(c) Compiler. The Ada ACCESS type which designate VMM objects are specified as accessing checkpointed data by means of the mark_release compiler PRAGMA [I-4, 10.2.2.3]. Such types are not subject to garbage collection or explicit storage reclamation that depends on the type of the object designated. Values for these types may thus be generated by unchecked conversions of an integer type suitable for address computations without interacting badly with Ada run-time storage management. The VMM representation analyzer has a direct interface with the compiler in that it is partly constructed from the LEXSYN and SEM compiler phases [I-4, 3.2.2 and 3.2.3].

3.2 Detailed Functional Requirements

3.2.1 Text Parsing

This section discusses the parser and lexer generators. The corresponding parser and lexer are described as part of LEXSYN [I-4 3.2.2]. Figure 3-1 provides a block diagram of the two generators.

3.2.1.1 LR Parser Generator

(a) Characteristics. LR, the parser generator developed at Lawrence Livermore Laboratory [Ref N-3], serves as the basis for the MAPSE parser generator. LR was used to build the Ada parser for the compiler Intermetrics is implementing for DARPA. Thus LR is sufficient to handle Ada. A JOVIAL compiler project at Intermetrics has created an enhanced version of the LR parser skeleton displaying improved error recovery. This is exactly the kind of error recovery required by LEXSYN [I-4, 3.2.2.2 (b)].

(b) Input. The parser generator has a single input file, consisting of an LR(1) grammar expressed in BNF. (This grammar for Ada already exists as a by-product of the DARPA Ada compiler project.) Each grammar production is augmented by actions to be taken when the corresponding reduce transition occurs during parsing. The actions employed for LEXSYN deal with the construction of pieces of the AST (abstract Syntax Tree) for the program.

(c) Processing. LR utilizes the efficient grammar analysis algorithm of Pager [N-4]. The details of constructing LR(1) parse tables can be found in [Ref N-3], [Ref N-4]. In summary, the full characteristic finite-state machine with one-symbol look-ahead is constructed, but states are merged when equivalent as long as this does not introduce inadequacies.

(d) Outputs. Three output files are created by the parser generator.

The Listing file contains the following information:

1. errors in the grammar;
2. terminals and non-terminals of the grammar, with internal numbers assigned by the generator;
3. pretty printed grammar;
4. LR properties of the grammar (optional);
5. complete state-transition description of parser (optional)

The latter two listing sections (4 and 5) are useful for debugging a grammar.

The Token Table file contains the data in (2) above, for grammar terminals, in tabular form. The terminals are exactly the tokens that are accepted by the lexer. When a terminal is recognized by the lexer, the internal number assigned by the parser is returned. The Token Table communicates these numbers to the lexer generator.

The Parse Table file contains an encoding of the states and transitions of the LR parser, plus the actions associated with productions. This file and the Token Table file are combined with the parser skeleton to form the parser.

3.2.1.2 Lexer Generator

(a) Inputs. The lexer generator has two input files: the Token Table file created by the parser generator (3.2.1.1 (d)), and a file consisting of the grammar for the tokens expressed as regular expressions. Actions are associated with the regular expressions.

(b) Processing. There are two classes of tokens, which are treated somewhat differently by the resulting lexical analyzer.

1. Keywords and special operators (e.g., "BEGIN", "+") are each distinct tokens and uniquely stand for themselves. The lexer returns only the internal token number to the parser.

2. Other terminals (e.g., <identifier>, <real literal>) represent a whole family of character strings in the input stream. The lexer returns both a token number and a value; e.g., X or 3.14159.

The parser generator easily distinguishes between these two classes and passes this information on via the Token Table. (Class 1 terminals appear literally in the LR(1) grammar; class 2 terminals appear as non-terminals which have no defining production in the LR(1) grammar.)

In the resulting lexer, the class 1 terminals are kept in a hash table. One action that can be specified is to look up the current input string in the hash table. If a match is found, then the corresponding token number is returned. In the case of no match, a specified default token is returned. For example, this action would be used for the regular expression representing "character optionally followed by characters and digits optionally containing interspersed underscores." If such a string is not a keyword, then it is recognized as an <identifier>.

This hash table scheme for class 1 terminals has several advantages.

- It is not necessary to create regular expressions for all of the keywords and operators.

- The lexical analyzer does not have numerous single-character transition states in order to spell out each class 1 terminal.

- The hash function for the keywords can be optimized by the lexer generator, making keyword recognition efficient.

The more straightforward lexical action is to return the specified class 2 terminal when the corresponding regular expression is recognized.

(c) Outputs. Two files are output from the lexer generator.

The Listing file contains a "prettied" version of the tokens and the regular expressions, and the state-transition description of the finite-state automaton.

The Lex Table file consists of the keyword hash table, and the table of states and transitions. These tables are combined with the lexer skeleton to create the le [illegible]

3.2.2 Data Management (Virtual Memory Methodology)

3.2.2.1 VMM Objects, Database Objects, and Domains

VMM provides for the definition, creation, access, manipulation, and storage (within the KAPSE database) of VMM objects. A VMM object may either be a data aggregate with abstract properties defined by the VMM implementation (arrays, lists, sets, strings) or else it may be a virtual record with components defined by the user of the VMM implementation (i.e., a MAPSE tool builder).

A virtual record is realized in Ada as an object of some record type with a single discriminant of some enumeration type; the discriminant is used only for case selection of components and not for constraints on component types. Further, the component types have statically determined sizes and contain no Ada ACCESS types or types for which assignment is not available. This record type thus defines a set of virtual record subtypes, where each value of the discriminant names a virtual record subtype with a statically determined size and representation; since it contains no ACCESS, TASK, or LIMITED PRIVATE types, objects of the type can be safely written to external files (i.e., KAPSE database objects accessed through package INPUT_OUTPUT) and read by subsequent program activations. The central function of the VMM package involves the creation of VMM objects within external files and the support of direct references from one VMM object to another in an efficient manner whether the VMM objects reside in the same or different external files. (Note: the term external file will be used in these sections to designate a KAPSE database object identified as a STRING to package INPUT_OUTPUT, while the term object will be used in the Ada sense; the term VMM object will be used to designate VMM aggregates and virtual records which are allocated within external files by VMM operations and are identified by VMM locators).

VMM operations which involve VMM objects must have a method for locating those objects within external files and establishing the subtype of virtual records with respect to the proper virtual record type. The method is that of defining the VMM locator type as a sub-domain number and a position within that sub-domain. A locator has meaning only with respect to a dynamically created virtual memory domain: the VMM implementation maintains, during the course of a program activation, an association between a virtual memory domain and a number of sub-domains which are defined as an association of a sub-domain number, an external file name, the name of a virtual record type, and a mode of operation.

(a) Domain Characteristics. In order to access VMM objects, a program must first create a domain. The function new_domain returns a value of the private type VMM_domain, which identifies a domain that is initially empty (has no sub-domains). Sub-domains can be added to this domain by calling the procedure new_sub_domain, with parameters:

1. Domain

2. Sub-domain number (VMM-defined integer type)

3. Name of external file

4. Name of virtual record type

5. Mode of operation (read, update, extend, create)

This procedure only establishes the sub-domain number as one that may be used to identify a sub-domain within the domain, and provides the information needed to access an external file when the sub-domain is referenced. If the sub-domain is never referenced, no database access occurs. If the sub-domain number is already available to the domain, the exception sub_domain_conflict is raised.

The name of the virtual record type is a value of an enumeration type defined in the VMM access package produced by the representation analyzer. This enumeration names the variant record types, each of which defines the kind of virtual records that may occur in a given sub-domain. This should not be confused with the enumeration types which are used for the discriminants of the variant records. For example, in the following fragment of such a package, the enumeration literals "tree" and "person" identify virtual record types, while "male" and "female" identify virtual record sub-types of the virtual record type "person". The virtual record types VMM_TRANSLATION and VREC_DESCRIPTION are defined by the VMM implementation and are always included in the VREC_types enumeration by the representation analyzer.

```
WITH VMM_implementation_pkg;
PACKAGE VMM_domain_pkg IS
        .
        .
        .
  PACKAGE tree_pkg IS

    TYPE tree_kind IS (t_access,t_derived);

    TYPE tree_record (kind:tree_kind) IS
        RECORD
          source_position:src_pos_type;
          CASE kind IS
                WHEN t_access => size:storage_integer;
                WHEN t_derived => type_spec:VREC_locator;
          END CASE;
        END RECORD;
    TYPE accessor IS ACCESS tree_record;
END tree_pkg;
         PACKAGE person_pkg IS

           TYPE person_kind IS (male,female);

           TYPE person_record (kind:person_kind) IS
               RECORD
                 CASE kind IS
                       WHEN male => bearded:boolean;
                       WHEN female => hairstyle:style_enum;
                END CASE;

              END RECORD;

          TYPE accessor IS ACCESS person_record;

      END person_pkg;


      TYPE VREC_types IS (VMM_TRANSLATION, VREC_DESCRIPTION,
                                              tree,person);

                  .

                  .

                  .

      END VMM_domain_pkg;
```

While a domain may contain virtual records of various virtual record types, a given sub-domain may only contain virtual records of the same virtual record type (although they may be of different sub-types).

Thus, a virtual memory domain is characterized by:

1. The set of virtual record types that are known within it. These types are defined by the tool builder and processed by the representation analyzer to produce a single Ada PACKAGE which includes the tool builder's type definitions and adds further information such as the enumeration of virtual record types just described. This characteristic is static.

2. The set of sub-domains of which the domain is composed. This characteristic is dynamic, in the sense that sub-domains are added to and removed from a domain by the execution of procedures, subject to the constraint that the virtual record type of each sub-domain is known in the domain. The set of sub-domains which compose a domain defines a virtual address space in terms of VMM locators: each locator identifies a sub-domain number and a position within that sub-domain.

(b) Sub-domain Characteristics. A sub-domain is identified by a domain and a sub-domain number. A sub-domain number is either present or absent with respect to a domain. Each sub-domain number present in a domain identifies an external file, a virtual record type, and a mode of operation; it provides access to a virtual address space in terms of positions within the external file. The external file is accessed using a single instantiation of the generic package INPUT_OUTPUT for elements of a record type defined by the VMM implementation as a VMM page: when the mode of operation for the sub-domain is read-only, access is through an object of type IN_FILE, and otherwise through an object of type INOUT_FILE.

The position-number of each element in the file is known as the page number. Read and write operations move the contents of a VMM page between the external file and a page buffer. Page buffers are assigned to the external files on a demand basis from a common pool of buffers allocated from the heap and shared by all sub-domains present in a program. Demand for a page buffer occurs when a page number within an external file is referenced, and that page number is not resident in a buffer currently assigned to the file. When demand occurs and all buffers in the common pool are already assigned, a buffer is selected for reassignment. If the contents of the page resident in that buffer has been modified since the buffer was assigned, the page is first written to its external file at the position indicated by its page number, and then the buffer is reassigned to the page of the external file that was referenced.

When a VMM object is created within a sub-domain, sufficient space for the object is located within the pages already assigned to its external file, or else a new page is created with a page number one greater than the largest page number assigned to the file. In either case, the space is reserved within a page and is designated by a VMM locator which is constructed from the sub-domain number which identifies the external file, the page number, and a position within the page. The position within the page is expressed as a VMM-defined integer type. When a sub-domain is added to a domain, its external file, virtual record type, and mode may be identified explicitly (as in procedure new_sub_domain) or a reference to an existing sub-domain may be provided instead, creating an alternate sub-domain number to access the same external file. The use of an alternate sub-domain number is restricted to contexts in which locators which use the alternate number are dereferenced: no VMM objects can be allocated using the alternate number. The purpose of using an alternate number is to allow the automatic translation of VMM locators which reference a given sub-domain into the corresponding VMM locators for a different sub-domain (which is a new version of the first sub-domain).

The automatic translation is specified by creating a translation sub-domain which contains a VMM set of locator associations; the set has membership testing based on locators for objects in the first sub-domain and associated values which are locators for objects in the new version. This translation sub-domain is created by comparing the two versions, and since all three are then present in a domain at the same time, they all have distinct sub-domain numbers. Once the translation sub-domain has been created, the original sub-domain is removed and the translation sub-domain is given an alternate number which is the same as that which identified the original sub-domain. When a VMM locator refers to the translation sub-domain through the alternate number, the translation is applied by looking up the locator in the set, and then using the new locator value for the reference. When a locator refers to the translation sub-domain through its original number, no translation is performed; thus the VMM locators used within the translation sub-domain to build the set are not "special". If it happens that for every member in the translation set, the translation changes only the sub-domain number in the locator (e.g., the new version was nearly identical to the original sub-domain) then both the original sub-domain and the translation sub-domain can be removed and the new version given an alternate number the same as the original's. When a locator refers to the new version through the alternate number, the fact that the sub-domain is not a translation sub-domain causes a new locator to be constructed by replacing the alternate number by the new version's orginal number.

When a sub-domain number is not an alternate number for a sub-domain, it is said to be the identity sub-domain for the external file it identifies. An identity sub-domain number is assigned to an external file by procedure new_sub_domain. When an identity sub-domain is first referenced, the external file is initialized with the number assigned (for create mode), or else the number from the external file is compared to the number assigned (for any other mode), raising the exception "wrong_sub_domain" if they do not match.

Each identify sub-domain has a single distinguished VMM locator, known as its root locator, which may be explicitly set or examined by VMM operations. Since all operations on VMM objects require a VMM locator to identify the object, the root locator generally identifies some object in the sub-domain which provides linkage to all the other objects in that sub-domain. However, since objects in one sub-domain may be referenced from another, this is not necessarily so: one sub-domain might serve as a directory for other sub-domains, in which case the directory sub-domain might be the only one with a non-null root locator. Further, a sub-domain which is temporary (i.e., its external file is created and deleted within a program activation) can be accessed by VMM locators held in program variables and need not provide for accessibility of its objects across program activations. In summary, a sub-domain is characterized within a domain by:

1. sub-domain number;

2. external file of VMM pages;

3. virtual record type;

4. mode of operation;

5. set of pages defined for placement in the external file;

6. set of buffers assigned to hold resident pages;

7. root locator.

When the sub-domain is not an identity sub-domain, then items (2) through (7) are represented by the sub-domain number of an identity sub-domain.

An external file that contains VMM data objects, but which is not seen in the context of an active sub-domain, is characterized by:

1. an identity sub-domain number;

2. a virtual record type;

3. a root locator;

4. a set of pages.

3.2.2.2 VMM Locators and Dereferencing

The previous section identified the functionality of VMM locators with respect to VMM domains and sub-domains. In particular, it described a translation property of VMM locators which reference alternate sub-domain numbers.

When a locator is dereferenced, the first step is always to examine the characteristics of the sub-domain it references and perform any indicated translation. When the sub-domain number is not an identity sub-domain, a translation is performed in one of two ways, depending on the virtual record type of the identity sub-domain that the referenced sub-domain is linked to.

1. If it is VMM_TRANSLATION, then the VMM locator is used as the identifying value for a find_member operation performed on the locator association set identified by the root locator of the identity sub-domain, and the value returned is used as the locator to be dereferenced. (If the returned value also references an alternate sub-domain number, the translation is repeated.)

2. If the virtual record type is not VMM_TRANSLATION, then a VMM locator is constructed which is the same as the input locator, except that the sub-domain number is replaced by the number of the identity sub-domain.

In either case, the translation always results in a VMM locator which references an identity sub-domain number. The remainder of this section then is concerned primarily with locators that reference identity sub-domains (which are the actual repositories for VMM data structures).

(a) Locator Model. A VMM locator is the only means of consistently designating a VMM object. A locator is, in many ways, similar to an Ada ACCESS value: it is a "typed pointer" with values generated by allocation operations; and a distinguished null value which designates no object at all. Locators within virtual records are used to implement data structures that can be conceptualized as attributed directed graphs in the same way that Ada ACCESS values within Ada RECORD objects would be used. Functional differences between VMM locators and Ada ACCESS values are:

1. A VMM locator value generated by an allocation during a program activation can be written to an external file and then be read by a subsequent program activation and still be guaranteed to designate the same VMM object. By contrast, an Ada ACCESS value is only defined within the context of the program activation which performed the allocation.

2. The addressing range of a VMM locator is defined by the implementation of the VMM package, and is not dependent on host machine characteristics or on the size of the run-time heap available to the program activation.

Considering only the first distinction, the functionality of VMM locators could be achieved by defining conversion operations between external files of VMM objects with directed edges represented by VMM locators, and internal collections of Ada objects with directed edges represented by ACCESS values. This kind of approach has been used in some test-bed compiler implementations, where the external file representation is a sequential text file and each compiler phase reads, and converts to internal form, the output from the previous phase, operates on the internal form, and then writes a text file representation to be read by the next phase [N-6, N-7]. Another approach to achieving the first functionality avoids using ACCESS values at all; instead, graph nodes are represented by elements of a single array of variant records, with directed edges stored as array indices [N-2, N-5]. Although this offers the advantage that no conversion processing is needed when reading or writing an external file, memory space utilization becomes a critical issue in the design of the record variants since the array elements will be allocated the space required for the largest variant: optimal utilization occurs when each variant is the same size. In addition, using array indices to implement pointers results in a loss of clarity and run-time efficiency, motivating the design of ACCESS types in the Ada language in the first place [N-8].

The second functional aspect of VMM locators is not addressed by the original implementation strategies for either TCOL or AIDA : both rely on sufficient memory resources to contain the internal representations for all nodes that are used during a graph traversal.

While the direct addressability of both the VM/370 and OS/32 systems is sufficient to accommodate intermediate representations for useful Ada programs, an implementation of the compiler and separate compilation facility that assumed adequate memory to access all nodes as Ada objects could not be rehosted to a system with a smaller address space without seriously impacting processing capacity.

Further, experience with a separate compilation facility for a Pascal dialect has shown that even the 24-bit addressing range of VM/370 imposes compile-time limits that can be exceeded by large applications. Therefore, the model for VMM locators requires, as an integral part of their use, a discipline which provides nearly complete independence from memory and addressing constraints imposed by the host machine. Not only does this requirement establish the portability of MAPSE tools to smaller machines, but it removes the absolute limits on processing capacity inherent in any program which relies on memory addressing to access an internal database. The actual addressing range for locators is an implementation choice; one possibility considered a candidate is to implement locators to address 4096 sub-domains, each with up to 4096 pages, and each page containing 4096 bytes addressed as 256 16-byte units. Such a locator value could be represented in 32 bits and would permit up to 16 megabytes of data in each sub-domain.

(b) Mapping Locators to Objects. The VMM strategy uses locators which have the same representation internally within an executing program as they do on an external file - in that sense similar to the AIDA approach using array indices. However, instead of implementing each element of the data structure as an element of an Ada array and using an array index to provide access, the VMM approach implements a VMM object as a block of storage units within a page buffer. The ADDRESS pre-defined attribute is used to convert an ACCESS value for the buffer to an implementation-defined integer type to which an offset is added before being converted to an ACCESS value of a type appropriate to the VMM object being accessed (by means of UNCHECKED_CONVERSION). The advantages of this approach are basically two-fold:

1. The relative sizes of the record variants to be accessed do not affect the efficiency of storage utilization: the number of VMM objects accessible within a buffer is determined by the minimum space required for each object and not by a worst-case fit based on the largest variant.

2. Once an ACCESS value is computed for a VMM object, references to its components do not require repeated indexing operations.

Recalling that a VMM locator encodes a sub-domain number, a page number, and a position, the mapping from locator value to ACCESS value is accomplished in two steps:

1. Locate a page buffer assigned to the sub-domain which contains the specified page, returning an ACCESS value for the buffer.

2. Compute the ACCESS value for the object as object_type (buffer.ALL'ADDRESS + position) where object_type is an instantiation of the generic function UNCHECKED_CONVERSION with input type an implementation-defined integer type and result type an ACCESS type for a VMM object.

In terms of the VMM implementation, a single procedure, locate, takes IN parameters consisting of the domain and the locator value and produces OUT parameters consisting of the ACCESS value for the buffer and the value Buffer.ALL'ADDRESS + position. The locate procedure is defined in the visible part of the VMM implementation package, and is thus visible to the VMM access package produced by the representation analyzer (since the analyzer places its name in the WITH clause for the access package). The tool using the VMM access package does not normally have the implementation package in its visibility list, and so it only makes use of the locate procedure by means of the dereferencing functions defined in the access package. These functions return values of an unconstrained ACCESS type for a virtual record type. Thus, the instantiation and use of UNCHECKED_CONVERSION is limited to the package (body) generated by the representation analyzer and is not used or seen by the tool which uses VMM. Furthermore, the VMM implementation package verifies the virtual record type for each sub-domain (in terms of the IMAGE, or character-string representation, of the virtual record type enumeral) once, when the external file is first accessed. The dereferencing functions generated by the representation analyzer for each virtual record type then use the ACCESS value for the page buffer (returned by locate) to verify that the "integer" being converted to an ACCESS value is in fact the address of a VMM object that was created in a sub-domain specified to contain objects of that same virtual record type. Thus, the use of UNCHECKED_CONVERSION by the VMM access package to achieve efficient storage utilization does not compromise Ada type safety. Subsequent component selection using the "converted" ACCESS value is again subject to the normal discriminant verification applied at run-time to record variants designated by unconstrained ACCESS values.

(c) Dereferencing. The process of obtaining an Ada ACCESS value in order to perform operations on a VMM object designated by a VMM locator is called locator dereferencing. Since locators define an address space which is not constrained by the addressing and memory-size limits of the run-time model for Ada objects, it is clear that the memory available for Ada objects will, in general, only be able to represent a subset of the VMM objects accessible to a program through locators. The subset of VMM objects resident in memory at any given time must include at least those objects that are named by ACCESS values in Ada statements. However, a dereference operation which identifies a VMM object that is not resident in memory at that time, may need to reuse space previously assigned to a VMM object, causing the ACCESS value computed for that object to become invalid: dereferencing drives the demand paging mechanism. Since the algorithm that determines which resident objects (i.e., which page buffer) will be replaced during a dereference operation cannot predict the future pattern of use for previously computed ACCESS values, it is always possible that the replaced objects would include those which were about to be referenced again. In the absence of further interaction with the dereferencing and underlying paging mechanism, only the most recently computed ACCESS value could be considered valid at a given point in a program execution since the ACCESS values invalidated by that computation (if any) cannot be determined by the tool builder.

Thus, every use of data held within a VMM object could require a locator dereference to obtain an ACCESS value guaranteed to be the most recently computed value: dereferencing can be viewed as being analogous to a "load" instruction on a single accumulator machine (actually, rather than "load" and "store", VMM defines "load read-only" and "load for update" operations with an implicit "store" carried out when a value loaded by the second form is about to be overwritten). Using the analogy, it should be clear that the performance of a program using VMM depends critically on the speed of the dereference operation, and that providing a capability analogous to a multiple-register architecture would allow significant performance enhancements to be made. VMM defines a method for reducing the cost of dereference operations in certain cases by splitting the operation into two separable parts called pointer computation and pointer dereference; a capability called dereference locking allows a program to force a VMM object to remain resident and accessible by the same ACCESS value throughout a specified region of program execution, effectively allowing full access to more than one VMM object at a time (a multiple-register architecture). These operations are essentially optimization techniques that can be employed by the tool builder using VMM; they do not change the basic model for locators and dereferencing.

POINTER COMPUTATION: The mapping from a VMM locator to an ACCESS value was described as a two-step process: (1) locate a page buffer containing the required page from the specified sub-domain and (2) perform the address arithmetic. Of the two steps, the first is much more costly. Even when the desired page is already resident in a buffer and no I/O operations are required, finding that buffer from the locator value alone implies some type of indexing or searching operation. Given the locator realization considered earlier (supporting 4096 sub-domains, each with up to 4096 pages), it is clear that direct indexing using the sub-domain number and page number would not be possible, requiring some type of search. While the use of simple hashing techniques can make the search time quite short, just computing a hash takes longer than the time needed to access the data once the buffer is found. A pointer computation converts a VMM locator (in the context of a VMM domain) to a form that already has had translations applied (i.e., it references an identity sub-domain) and which "remembers" intermediate results from the last time it was dereferenced. In particular, a VMM pointer contains:

1. VMM domain value;

2. VMM locator value;

3. ACCESS value for a page buffer;

4. An address within the page buffer (implementation-defined integer);

5. A timestamp (implementation-defined integer).

The first four values are obtained by performing any necessary translation on the input locator value and then invoking procedure locate to compute the third and fourth values. The timestamp value is simply copied from the page buffer.

Each page buffer has a timestamp component which is updated by the page buffer management routines whenever the buffer is reassigned to a different page or deassigned from active use (e.g., when a sub-domain is removed). The update consists of incrementing a single counter which is maintained for the entire pool of page buffers, and then copying the incremented value to the particular page buffer being reassigned. The range of the counter is such that it could never overflow during the course of a single program activation; it is initialized once (by elaboration of the library unit for the VMM implementation package) and never reset. (A 32-bit integer is adequate for this purpose). The result is that the timestamp value recorded in the buffer designated by a VMM pointer will equal the timestamp value recorded in the pointer itself if and only if the same page has remained resident in that buffer since the pointer was computed.

POINTER DEREFERENCING: Once a VMM pointer has been computed from a VMM locator, dereferencing the pointer to obtain an ACCESS value can be accomplished with no hash computation or searching at all as long as the VMM object remains resident. If the object has not remained continuously resident since the pointer was computed, the cost of the dereference is no greater than the cost of dereferencing the original locator, and may be less (i.e., if the original locator required translation). Furthermore, the processing is so simple that pointer dereference functions can be specified as inline (via the language-defined PRAGMA), with the result that no procedure calls are required when the object is resident.

DEREFERENCE LOCKING: When a dereference is locked, the VMM object that is designated by the dereference is forced (by the buffer management routines) to remain resident in the same buffer, and thus remain accessible by the same ACCESS value for as long as the lock remains in effect. This capability allows a program to safely designate more than one VMM object by an ACCESS value; at any point in a program, the VMM objects that can be designated by ACCESS values are those that are locked plus the most recently dereferenced value.

The way that a dereference is locked depends on whether locator or pointer dereferencing is used. A locator dereference is locked by specifying "lock=>true" as a parameter to deref_lctr; the ACCESS value returned will then be locked. A pointer dereference is locked by invoking procedure lock_ptr with the VMM pointer as parameter; if the VMM object is not shown to be resident by the pointer, the exception "pointer_not_resident" is raised. An advantage of locking pointer dereferences is that the decision to lock can be made after the content of the VMM object has been examined.

Both kinds of dereferences are unlocked by a checkpointing mechanism. Procedure mark produces an OUT parameter of type VMM_mark; procedure release takes an IN parameter of the same type. A checkpoint is established by invoking mark and identified by the value it returns. All dereferences which become locked after a particular checkpoint is established are unlocked by invoking release with the same checkpoint value or a checkpoint value which was established earlier. Pointer dereferences can also be unlocked on an individual basis using procedure unlock_ptr.

3.2.2.3 VMM Data Types

VMM objects are typed objects in the same sense that Ada objects are typed. In general, a VMM object is designated by a typed VMM locator, where the type of the locator identifies the object as either:

1. a virtual record object with components defined by the user of the VMM implementation by means of virtual record type declarations; or

2. an object representing a particular data abstraction directly supported by the VMM implementation.

All such locator types are derived from a single locator type defined by the VMM implementation package, for which the locate procedure is defined (see Section 3.2.2.2(b)).

The only operations available to the user of VMM are those defined in the visible part of the VMM access package generated by the representation analyzer. That access package contains dereferencing operations only for the locator type that designates virtual records. For the other locator types, only the appropriate operations for each abstraction are provided. Thus, while VMM locators can be seen as inherently "typeless" at the lowest level of implementation, indicating a position within an external file to be translated to a memory address within a buffer, derived types are used to establish compile-time verification of consistent locator usage. Further run-time checks are applied by both the compiled code (e.g. disciminant verification) and by the VMM implementation (e.g. virtual record type verification for each sub-domain).

(a) Virtual Record Type. A program examines and modifies a virtual record by dereferencing its locator and then operating on the components of the Ada record object designated by the ACCESS value so obtained. Although a MAPSE tool will generally use the type encapsulation facilities of Ada to operate on virtual records as attributed nodes of threaded lists, trees, dags (directed acyclic graphs), or other structural abstractions, the VMM access package is not "aware" of this higher level of abstraction. Rather, the VMM access package is the means for implementing these abstractions. Thus, the Diana abstract data type is implemented in terms of virtual records, with the Diana implementation performing dereference operations unseen by the tool operating through the Diana package. This is not to say that virtual records are crude low-level data containers. Besides simple scalar values or references to other virtual records, virtual record components may contain statically-constrained arrays, nested records, variable-length strings, or references to objects of an abstract data type supported by VMM.

For each virtual record type, there are two dereferencing functions and one allocation function generated by the representation analyzer and specified in the visible part of the access package it produces. One dereference function operates on VMM locators (for virtual records), the other on VMM pointers. The allocation function returns a VMM pointer rather than a locator; very often an allocation is shortly followed by one or more dereferences (e.g. to assign initial values), and returning a pointer preserves the buffer ACCESS value computed during the allocation. If only the locator value is needed, the function ptr_to_lctr can be applied to the result at little cost, since it can be specified as an inline function that simply returns the locator component of its single pointer parameter. For example, the specifications generated for the "tree" virtual record type would be as follows:

```
     FUNCTION deref_lctr(vmd: VMM_domain;

                         lctr: VREC_locator;

                         lock: boolean:=false;

                         modify: boolean:=true) RETURN
                                                 tree_pkg.accessor;

     FUNCTION deref_ptr(ptr: VREC_pointer)  RETURN
                                                 tree_pkg.accessor;

     FUNCTION new_tree(vmd: VMM_domain;

                       vmsd: VMM_subdomain;

                     kind: tree_pkg.tree_kind) RETURN VREC_pointer;
```

An additional pair of procedures is generated for each virtual record type to support the reading and writing of both the human readable (VRN) and compressed binary representation of VMM data. The first procedure takes a string parameter that names a database object; it builds a symbolic description of the virtual record type within that database object. The symbolic description includes the (character string) name of each virtual record component applicable to each virtual record subtype, and a description of the type of each component sufficient to allow correct conversion between the internal value held within the component and a textual (VRN) representation of that value. The symbolic description is itself represented by VMM objects of a virtual record type defined by the VMM implementation as "VREC_DESCRIPTION". The database object containing the description of virtual record type is associated with an identitiy sub-domain of a VMM domain which is managed by the VMM implementation independently from the domains created by the VMM user.

The second procedure of the pair also takes an external filename parameter, but instead of building a virtual record type description, it accesses a description previously built within that object by execution of the first procedure. The specifications for the "tree" virtual record type would be

```
PROCEDURE build_tree_description(filename:string);

PROCEDURE get_tree_description(filename:string);
```

Besides these functions and procedures, whose bodies are generated specifically for each virtual record type by the representation analyzer, there are a number of operations that are applicable to virtual records but use the same specification and body for locators designating objects of different virtual record types. The specifications and bodies for such operations are generally part of the VMM implementation package; the VMM access package produced by the representation analyzer makes them available to other compilation units by renaming declarations (compilation units using VMM mention only the access package and not the implementation package in their context specifications). Among these operations are the following:

```
PROCEDURE set_root(vmd: VMM_domain;

                   vmsd: VMM_subdomain;

                   root: VREC_locator);

     -- set root locator value for a sub-domain

FUNCTION get_root(vmd: VMM_domain;

                 vmsd: VMM_subdomain) RETURN VREC_locator;

     -- obtain root locator value for a sub-domain

FUNCTION next_vrec(vmd: VMM_domain;

                   vmsd: VMM_subdomain;

                   this_vrec: VMM_cell_locator) RETURN

                                          VMM_cell_locator;

     -- Iterate overall virtual records within a sub-domain.

     -- When this_vrec is VMM_null, the first virtual record is

     -- obtained.
     -- When this_vrec identifies the last virtual record
     -- created VMM_null is returned.
FUNCTION cell_value(vmd: VMM_domain;
                  cell: VMM_cell_locator) RETURN VREC_locator;
     -- Convert the VMM_cell_locator returned by next_vrec to a
     -- VREC_locator.
PROCEDURE delete_vrec(vmd: VMM_domain;
                      vrec: VREC_locator);
     -- The specified virtual record will no longer be found by
     -- next_vrec, and subsequent use of the locator for it is
     -- erroneous.
PROCEDURE vrn_output(vmd: VMM_domain;
                     file: TEXT_IO.OUT_FILE);
PROCEDURE vrn_output(vmd: VMM_domain;
                     vmsd: VMM_subdomain;
                     file: TEXT_IO.OUT_FILE);
PROCEDURE vrn_output(vmd: VMM_domain;
                     vrec: VREC_locator;
                     file: TEXT_IO.OUT_FILE);
```

```
          -- Output an entire domain, a sub-domain, or a single
          -- virtual record in virtual record notation.
     PROCEDURE compressed_output(vmd:  VMM_domain;
                                 file:  TEXT_IO.OUT_FILE);
          -- Output a domain in compressed binary form.  No smaller
          -- granularity  is   provided  for   compressed   form.
     PROCEDURE vrn_input(vmd:  VMM_domain;
                         file:  TEXT_IO.IN_FILE);
     PROCEDURE compressed_input(vmd: VMM_domain;
                                file:   TEXT_IO.IN_FILE);
          -- Input a domain in virtual record notation or
          -- compressed binary form.  No smaller granularity is
          -- provided for input operations.
```

(b) Supported Data Abstractions. VMM directly supports abstract (or encapsulated) data types for doubly-linked lists (chains), arrays, uncounted sets, and variable-length strings. Direct support means that operations on the data types are defined by the VMM implementation, and that there is a distinctive representation defined for the type in virtual record notation. The fact that these types are directly supported does not mean that other commonly used structures such as trees or dags cannot be defined as abstract data types; it simply means that the implementation of those types must be provided by the user of VMM in terms of the virtual record types on which they will operate, and that the human readable representation for objects of those types will show the structure by means of label references in virtual record components. The directly supported types are not inherently more efficient than such types could be implemented using virtual records defined by the tool builder.

The reason for supporting these particular abstractions has to do primarily with the advantage gained in having a simple "standard" human-readable representation for them. Lists, arrays, and sets have a natural representation as a sequence of elements which, with the possible exception of arrays, is much more comprehensible than a lower-level exposition of the directed reference structures used to implement them. A quoted string is more pleasant to look at than a sequence of separate characters, besides being able to represent clearly a zero-length string. While trees can be expressed in a somewhat more comprehensible form by nesting, indenting, or a more graphic means, the gain in clarity over a simple referential form is not nearly as great and quickly becomes lost when the size and depth of the structure approach a size that is likely to be encountered in actual applications, or when the form is adapted to include the more frequently encountered dag. Of course, supporting data abstractions which have a simple human-readable representation is not useful if the abstractions themselves are not useful. Extensive experience with compilers, linkers, separate compilation databases and other software development and support tools has shown the utility and effectiveness of the VMM-supported abstractions.

The following paragraphs briefly describe the functionality of each abstraction and provide Ada specifications for the operations applicable to each. In general, there is a group of operations whose specifications depend upon the type of elements organized by the abstraction and another group whose specifications are fixed. The first group is generated by the representation analyzer for each element type defined to be organized by that abstraction in a virtual record type definition. This is indicated by the notation element_pkg.typename, where element_pkg is considered to identify a package generated by the representation analyzer to contain declarations associated with a particular element type; operations are assumed to be overloaded on the various element types which use it. The second group of operations is generally defined in the VMM implementation package to operate on the parent type of locator for the particular data abstraction. These are made available in the VMM access package by renaming declarations.

ARRAYS: The array abstraction supported by VMM is similar to the Ada capability for dynamically-sized single-dimension arrays. However, VMM arrays are not simply mapped into corresponding Ada array objects since that would limit the size of a VMM array to the size of a page buffer. VMM arrays are indexed by natural numbers and have a lower bound of one.

Operations generated by the representation analyzer:

```
     FUNCTION new_array(vmd: VMM_domain;

                        vmsd: VMM_subdomain;

                        size: VMM_array_size)

                              RETURN element_pkg.array_locator;

     FUNCTION element_value(vmd: VMM_domain;

                            array_lctr: element_pkg.array_locator;

                            index: VMM_array_size)

                                   RETURN element_pkg.element_type;

     PROCEDURE element_value(vmd: VMM_domain;

                             array_lctr: element_pkg.array_locator;
```

```
                                   index: VMM_array_size;

                                   value: element_pkg.element_type);
```

Operations renamed from the implementation package:

```
     FUNCTION size(vmd: VMM_domain;

                   array_lctr: VMM_array_locator)RETURN
                                                     VMM_array_size;

     PROCEDURE delete(vmd: VMM_domain;

                      array_lctr: VMM_array_locator);
```

LISTS: VMM lists are modeled on doubly-linked circular chains. List operations deal with cell locators that may identify either a list cell (which represents a single element) or they may identify a list header (which represents a complete list). When a list is created, a list header is allocated which is linked to itself and its locator value is returned, with the result that subsequent operations treat it as designating an empty list. Both individual cells and list segments may be moved between lists.

Operations generated by the representation analyzer:

```
     FUNCTION new_list(vmd: VMM_domain;

                       vmsd: VMM_subdomain)

                               RETURN element_pkg.cell_locator;

     FUNCTION new_cell(vmd: VMM_domain;

                       vmsd: VMM_subdomain;

                       initial: element_pkg.element_type)

                              RETURN element_pkg.cell_locator;

     FUNCTION cell_value(vmd: VMM_domain;

                         cell: element_pkg.cell_locator)

                              RETURN element_pkg.element_type;

         -- exception if cell is a list header.

     PROCEDURE cell_value(vmd: VMM_domain;

                          cell: element_pkg.cell_locator;

                          value: element_pkg.element_type);
```

```
        -- exception if cell is list header.

   Operations renamed from the implementation package:
        FUNCTION next_cell(vmd: VMM_domain;
                           cell: VMM_cell_locator)
                                 RETURN VMM_cell_locator;
        -- VMM_null if last cell in list
        FUNCTION prev_cell(vmd: VMM_domain;
                           cell: VMM_cell_locator)
                                 RETURN VMM_cell_locator;
        -- VMM_null if first cell in list
        PROCEDURE before_cell(vmd: VMM_domain;
                              cell_A, cell_B,: VMM_cell_locator);
        -- Inserts cell_B before cell_A.
        -- If cell_A is a list header, cell_B becomes the last
        -- cell.
        -- If cell_B is a list header, all of its list cells are
        -- inserted before cell_A, leaving the cell_B list empty.
        -- Exception if cell_A is not in a list, cell_B is in a
        -- list and is not a list header, or if cell_B is cell_A's
        -- list header.
        PROCEDURE after_cell(vmd: VMM_domain;
                             cell_A, cell_B: VMM_cell_locator);
        -- Like before_cell only insertion is after cell_A.
        PROCEDURE remove_cell(vmd: VMM_domain;
                              cell: VMM_cell_locator);
        -- cell is removed from the list it is in (if any).
        FUNCTION tail(vmd: VMM_domain
                    vmsd: VMM_subdomain;
                    list: VMM_cell_locator;
                    cell: VMM_cell_locator)
                        RETURN VMM_cell_locator;
        -- Creates a new list in the specified sub-domain
        -- and adds to it all cells from list which follow
        -- cell, removing them from list.
    PROCEDURE delete(vmd: VMM_domain;
                      cell: VMM_cell_locator);
        -- Removes, and then deletes, cell.  If list header, then
        -- deletes all cells in list and header.
    FUNCTION size(vmd: VMM_domain;
                    list: VMM_cell_locator)
                        RETURN VMM_list_size;
        -- Number of cells in list.
```

SETS: VMM sets are modeled on uncounted sets with a somewhat different realization that depends on the membership testing algorithm specified when the set is created. The primary characteristic of sets is the speed of finding a member; insertions, deletions, and even iterations over all members are not necessarily simple operations.

The algorithm for membership testing is specified as a value of a VMM-defined enumeration type and is known as the equality type for the set. There are five basic equality types as follows:

1. Bit vector equality. Applicable only to sets with elements of a discrete type; the realization is a bit vector, as in Pascal sets.

2. Identity equality. Set members are equal if elements are the same; for scalar types the values must be equal, while for sets of VMM locators the designated objects must be the same object.

3. String equality. Applicable only to sets of VMM strings; strings are equal if they have the same length and their characters are the same and in the same order. Case of alphabetics is significant. Zero-length strings are equal.

4. Locator association equality. The realization is a set of locator associations, each member consisting of a key locator value and an associated locator value; identity equality is applied to the key locator value.

5. *Key-component equality. Applicable only to a set of* virtual record locators; all virtual records must be of the same virtual record type. A virtual record definition may specify that certain components belong to a particular key-component class. For any given virtual record subtype, there must be at most one component applicable to that subtype which belongs to a given key-component class. Membership testing is based on identity equality of the key component belonging to the class specified when the set was created, unless the key component is a VMM string, in which case string equality is used.

Key-component equality permits general associative addressing of virtual records; locator association equality is essentially a special case of this type of equality recognized for its general usefulness as well as its specific application to translation sub-domains.

Operations generated by the representation analyzer:

```
     FUNCTION new_set(vmd: VMM_domain;

                      vmsd: VMM_subdomain;

                      equality: VMM_equality;

                      class: natural;

                      size: VMM_set_size)

                            RETURN element_pkg.set_locator;

     FUNCTION member_value(vmd: VMM_domain;

                           member: VMM_member_locator)

                                   RETURN element_pkg.element_type;

     PROCEDURE add_member(vmd: VMM_domain;

                          set: element_pkg.set_locator;

                          value: element_pkg.element_type;

                          member: OUT VMM_member_locator);

          -- member returns VMM_null unless member to be added

          -- already present, in which case member returns that

          -- matching member.
```

```
FUNCTION find_member(vmd: VMM_domain);
                     set: element_pkg.set_locator;
                     key_value: element_pkg.key_type)
                                RETURN VMM_member_locator;
     -- key_type same as element_type except for key_component
     -- equality.  Returns VMM_null if member not found.
FUNCTION remove_member(vmd: VMM_domain;
                       set: element_pkg.set_locator;
                       key_value: element_pkg.key_type)
                                  RETURN boolean;
     -- true if member was in set
PROCEDURE add_member(vmd: VMM_domain;
                     set: VMM_assoc_set_locator;
                     key_values,
                     associated_value: VMM_locator;
                     member: OUT VMM_member_locator);
      -- Adds member to locator association set.
FUNCTION key_value(vmd: VMM_domain;
                   association: VMM_member_locator)
                                RETURN VMM_locator;
FUNCTION associated_value(vmd: VMM_domain;
                          association: VMM_member_locator)
                                RETURN VMM_locator;
PROCEDURE associated_value(vmd: VMM_domain;
                           association: VMM_member_locator)
                                RETURN VMM_locator;
PROCEDURE associated_value(vmd: VMM_domain;
                           association: VMM_member_locator;
                           value: VMM_locator);
```

```
          -- associated value can be modified
     FUNCTION copy(vmd: VMM_domain;
                   vmsd: VMM_subdomain;
                   set:  VMM_set_locator)
                        RETURN VMM_set_locator;
          -- makes a copy of the set in the specified sub-domain.
Operations renamed from the implementation package:
     FUNCTION next_member(vmd: VMM_domain;
                          set: VMM_set_locator;
                          member: VMM_set_member)
                                  RETURN VMM_set_member;
          -- Interates over members in arbitrary order
     FUNCTION size(vmd: VMM_domain;
                   set: VMM_set_locator)
                        RETURN VMM_set_size;
          -- number of members in the set.

     -- The following functions perform the indicated set
     -- operation on set1 and set2, leaving the result in set1.
     -- Functions return true only if set1 is modified by the
     -- operation.
     FUNCTION intersection(vmd: VMM_domain;
                           set1, set2: VMM_set_locator)
                                       RETURN boolean;
     FUNCTION union(vmd: VMM_domain;
                    set1, set2: VMM_set_locator)
                                RETURN boolean;
     FUNCTION difference(vmd: VMM_domain;
                         set1, set2: VMM_set_locator)
                                     RETURN boolean;
```

```
FUNCTION symmetric_diff(vmd: VMM_domain;
                        set1, set2: VMM_set_locator)
                                    RETURN boolean;

PROCEDURE delete(vmd: VMM_domain;
                 set: VMM_set_locator);
```

STRINGS: VMM variable-length strings are implemented much as in the TEXT_HANDLER package used as an example in [G-1, 7.6]. Manipulations are performed using Ada objects of tye character, type string and type VMM_text, with conversions from VMM string objects (designated by locators) to text objects and assignments from text objects to VMM string objects made explicitly when required. This makes effective use of overloading and the power of Ada functions for handling dynamically-sized objects, and also tends to reduce the total number of locator dereference. Note that the "&" operator could not be overloaded on VMM string objects since two values, domain and locator, are required to identity each object. Besides the allocate and delete operations there is only a single selection-function and update-procedure pair defined to operate on VMM string objects. No specifications need to be generated by the representation analyzer.

Operations renamed from the implementation package:

```
FUNCTION new_string(vmd: VMM_domain;
                    vmsd: VMM_subdomain;
                    initial: VMM_text)
                             RETURN VMM_string_locator;

FUNCTION value(vmd: VMM_domain;
               str: VMM_string_locator)
                    RETURN VMM_text;

PROCEDURE value(vmd: VMM_domain;
                str: VMM_string_locator;
                text: VMM_text);

PROCEDURE delete(vmd: VMM_domain;
                 str: VMM_string_locator);
```

3.2.2.4 Representation Analyzer

The representation analyzer is a MAPSE tool that reads one or more virtual record type definitions supplied as Ada package specifications, enforces the restrictions and conventions imposed by the VMM implementation, and generates a single Ada package specification and package body that defines an interface between the VMM implementation and any tool using those virtual record types. (See Figures 3-2, 3-3).

(a) Input. Input to the representation analyzer is a sequence of Ada package specifications. Each package specification defines a virtual record type and is subject to a number of restrictions and conventions in the way it is defined. The overall structure of each specification is as follows:

1. Each package mentions VMM_predefined_pkg in its WITH clause.

2. The name of the package satisfies a naming convention such that a name for the virtual record type can be constructed from it.

3. An enumeration type to be used for the discriminant of the virtual record type is declared with a type mark derived from the package name by a naming convention.

4. Types and sub-types are declared for use in the declarations of virtual record components. These declarations are subject to further conventions and restrictions described later.

5. A record type which defines the components of virtual records is declared with a type mark derived from the package name by a naming convention. This type declaration is subject to the following restrictions:

   a. It must have a single discriminant of the type identified in 3 above.

   b. The discriminant name may only be used in variant parts and not as a bound in an index constraint or a value in a discriminant constraint.

   c. The component declarations may only be of the form which specifies a subtype_indication, and that type mark must be either boolean, a name declared in the same package, or else a name declared in a preceeding virtual record type definition package. In the latter case, that other package must be mentioned in the WITH clause.

The type and subtype declarations for virtual record components are restricted to use a subset of the full type definition facilities of Ada. In particular:

1. No ACCESS, TASK, PRIVATE or fixed-point types may be defined.

2. Record types may not have discriminants.

3. Array types must be constrained and have only a single dimension.

A type mark used in the declaration of a virtual record component can be considered to fall in one of the following categories:

1. A simple type. This category includes pre-defined integer and float types (and types derived from them), boolean, and enumeration types.

2. A constructed type. This category includes records and one-dimensional arrays. It also includes constrained sub-types of the variable-length text string type VMM_text (defined in VMM_predefined_pkg).

3. A VMM locator type.

While the types in categories (1) and (2) are declared in the conventional manner in Ada, VMM locator types are declared by instantiating generic packages defined in VMM_predefined_pkg. One reason for this convention is that it allows the representation analyzer to associate additional information (the generic actual parameters) with virtual record components that contain VMM locators. This additional information includes such things as the type of elements in a VMM array list or set, the key-component class of a virtual record component, or a default size for a VMM set. Such information is needed to support conversions to and from human readable and compressed binary forms. Another reason for the convention is that it allows the tool builder to associate a specific derived locator type for a supported abstraction with the type of element organized by the abstraction. The complete specification of these generic packages and how they are used is an implementation issue, but the following example will convey the design approach.

Assume that a component of a virtual record is to contain a locator for a list of colors. Package VMM_predefined_pkg contains the following generic package specification for defining VMM lists:

```
GENERIC

    TYPE element IS PRIVATE;

PACKAGE list_gen IS

    TYPE cell_locator IS NEW VMM_cell_locator;

    SUBTYPE element_type IS element;
```

```
        FUNCTION new_list(vmd: VMM_domain;
                           vmd: VMM_subdomain)
                               RETURN cell_locator;

        -- etc, see Section 3.2.2.3 for other list operations

        -- generated by the representation analyzer.

    END list_gen;
```

The virtual record type definition package might then contain the following:

```
    TYPE color_type IS (red, green, blue);

    PACKAGE color_list IS NEW list_gen (color_type);
```

The declaration for a virtual record type describing wallpaper might then be written:

```
    TYPE wallpaper_record (kind:  wallpaper_kind) IS

        RECORD

           CASE kind IS

               WHEN solid => color: color_type;

               WHEN pattern => colors: color_list.cell_locator;

           END CASE;

        END RECORD;
```

From this the representation analyzer can build a description of the colors component that identifies it as designating a VMM list of colors with names of red, green, and blue. This enables the correct reading of the following virtual record notation (see 3.2.2.5 (a) and 10.1).

```
    500:  pattern [colors => <red blue>]
    501:  solid [color => green]
```

(b) Processing. Since the input to the representation analyzer is legal Ada, the bulk of the front end processing is done by invoking the LEXSYN, SEM, and (possibly) GENINST phases of the compiler. Since the generics used to specify virtual record type components are particularly simple (containing no procedure or function formal parameters) and in fact could be "known" by the representation analyzer (they are part of the VMM implementation, just like the representation analyzer itself), GENINST could be omitted and the represention analyzer would deal with the semantics of the generics it understands itself.

Which ever approach is taken to the generics, the Diana tree is examined to verify that the conventions and restrictions for specifying virtual record types have been observed. If not, messages are added to any messages already attached to the compilation unit by the compiler. If no errors are detected by the compiler phases, and no deviations from the VMM conventions are found, a new package specification and a package body are generated.

The package specification is a straightforward mechanical transformation of the input specifications, nesting them within a package specification named VMM_access_pkg, and adding further items such as:

1. Within each nested package add a type declaration named accessor which is an ACCESS type for the variant record type defined in it.

2. Outside the nested packages, define an enumeration type named VREC_types which lists the name of each virtual record type (obtained by a naming convention applied to each package name). The first two enumerals are always VMM_TRANSLATION and VMM_DESCRIPTION.

3. Add renaming declarations for entities declared in the VMM implementation package which need to be available to the user (e.g. the "fixed" operations described in Section 3.2.2.3). Also add renaming declarations outside the nested packages for procedures and functions declared by the generic instantiations inside the nested procedure, overloading them.

4. Add (overloaded) declarations for virtual record creation and dereferencing operations as well as other fixed declarations.

5. Add PRAGMAs where appropriate (e.g. arrays of boolean or character are packed, accessor types are checkpointed [I-4, 10.4]).

The package body generation requires:

1. Initializing Ada data structures in the generated package with the size of each virtual record subtype and the offset and type description of key components for each key-component class.

2. Generating the build_foo_description procedure (where foo in a virtual record type, see 3.2.2.3).

Both of these functions require imparting to the generated package certain characteristics of virtual record types, such as the size of an object of each subtype and the offset of each componet applicable to each subtype. This is done by generating expressions which use the predefined language attributes SIZE and POSITION, divorcing the representation analyzer from knowledge of the representation decisions that will be made by the compiler when the access package is compiled.

The names of the virtual record types and the names of all components applicable to each subtype are determined by examining the Diana symbol table.

(c) Output. The output from the representation analyzer consists of an error/summary listing and a package specification and body in source form. The error/summary listing contains any messages generated by the compiler phases used to process the input, as well as messages indicating any violations of the restrictions and conventions imposed by VMM. If there are no errors, a summary of the virtual record types may be optionally produced, showing for each virtual record subtype the names and types of components applicable to it.

The package specification output is named VMM_access_pkg, and it mentions VMM_implementation_pkg in its WITH clause. It contains the input specifications as nested packages and has other declarations as described in (b) above. The package body for VMM_access_pkg mentions UNCHECKED_CONVERSION in its WITH clause and supplies the bodies for those entities in the specification which require bodies.

3.2.2.5 External Forms

(a) Human-Readable (Virtual Record Notation). Virtual Record Notation (VRN) is a readable text representation used to describe VMM data structures. A MAPSE tool that operates on a virtual memory domain can convert the entire domain or a set of sub-domains to or from VRN, using operations declared in the VMM access package. VRN preserves the semantics of VMM data structures, including the distribution of virtual records among sub-domains and explicit sharing of data values. The syntax of VRN is derived from the syntax proposed for an external representation of Diana [reference] with some significant departures. These are motivated by the fact that VRN describes a data structure implementation rather than a data abstraction. The notation explicitly shows how the data structure is realized in terms of Ada records, database objects, and the set of data abstractions implemented by VMM. Thus, each VMM object is associated with the database object in which it is stored, and the VMM data abstractions used for aggregates of objects (arrays, lists, sets) are explicitly distinguished. On the other hand, the precise mapping between VRN and VMM objects is based on name associations determined by the representation analyzer; a representation can be changed by reading an instance of VRN using a different set of name associations than that used to create the instance. In such a case, discrepancies between the VRN being read and the specified mapping may be noted as "errors," even though the result may, in fact, be a useful representation conversion. A grammar for VRN is given in 10.1.

Referring to the grammar, it can be seen that a virtual memory domain is represented as a sequence of sub-domains. Each sub-domain first declares a list of sub-domain specifications that it will use in reference to virtual records. Each USE declaration associates a label for a sub-domain with components that identify a database object for that sub-domain and a virtual record description produced by the representation analyzer. The <use reference> which immediately follows the word DECLARE identifies the USE declaration that describes the sub-domain being represented. This is the "identity" sub-domain label, and it is assumed as a prefix to all record references (and record definitions) contained within the sub-domain. The form of components for a USE declaration that defines an identity sub-domain label must include both the name of the database object to be used for that sub-domain and the name of the virtual record type for the sub-domain. If a USE declaration contains a single component that is a <user reference>, then the label it defines identifies an alternate sub-domain that must not appear in any <use reference>.

Following the <declare part> are representations of the virtual records within the sub-domain (in the <record part>). The <record reference> which follows the word RECORD, identifies the root object of that sub-domain. Each <record def> represents a VMM object that is located within that sub-domain. The object is either a virtual record (with a structure defined by a virtual record description produced by the representation analyzer) or it is a form of data aggregate supported by the VMM implementation. The <label> on each object provides the means by which that object may be referenced from some other point within the <domain>; when referenced from within a different sub-domain, the second form of <record reference> is used, where the first <label> identifies the sub-domain in which the second <label> is to be resolved. Identification of the sub-domain is accomplished through the nearest preceding USE declaration bearing the same label. Within a domain, there must be at most one sub-domain that has an identity label naming a given database object, and all USE declarations that explicitly name the same database object must bear the same label. However, the same label may appear on more than one USE declaration and be associated with different database objects in different <declare part>s. For any given sub-domain label, only the most recent association is visible to a <record reference>.

A virtual record object is represented by an identifier followed by components enclosed in special brackets. The identifier names a discriminant value used in the virtual record description named by the "identity" USE declaration for the sub-domain. That discriminant value identifies the names and types of all components that may be part of the virtual record. Each component associates a value with a name. The name must be one of those component names that are applicable to the discriminant value, and the value must be compatible with the type defined for that component in the virtual record description. While the discriminant value restricts the names and types of components that may appear within the record's delimiting brackets, it does not require that each possible component be explicitly represented. Those components that have internally the value they are assigned when a virtual record is created (e.g., the value represented by binary zeroes) need not appear in the external representation.

(b) Compressed Binary Form. The compressed binary form of VMM data structures is semantically equivalent to the human-readable Virtual Record Notation for a VMM domain. The binary coding of data is accomplished by using the representation for the pre-defined type CHARACTER (which is specified in package STANDARD and is the same for all Ada implementations) to simulate an undifferentiated stream of bits. The bit stream is read from and written to Ada text files (defined in package TEXT_IO) using the read and write operations they derive from INPUT_OUTPUT; the get and put operation of package TEXT_IO are not used since these files have no line structure. The I/O is performed on seven-bit segments of the stream, where each segment is represented on the file by the character value CHARACTER'VAL(segment).

Space compression is achieved in two ways:

1. At the start of a compressed binary representation is a fixed-length, fixed-format prologue which specifies the length in bits of various control fields that are used to structure the remainder of the representation. Among these control fields are structuring identifiers that identify the logical elements of the representation, and length specifiers which permit each occurrence of a numeric value to be represented by the minimum number of bits required to express its magnitude. Thus, structuring identifiers play the role of the various terminal symbols of the VRN grammar. Length specifiers afford the same kind of "space compression" achieved when numeric values are represented textually with leading zeroes suppressed, except that the binary compression avoids the expansion introduced by the textual representation itself.

2. At a higher level, space is compressed by eliminating the representation of symbolic names for identifiers. Following the fixed-length prologue is a table of all identifier names needed to express the data structure in VRN; in the data structure representations which follow the table, a reference to an identifier is coded as an index into the table. In this way, the exact semantics of name association used in VRN are preserved.

Because this type of binary representation depends only on the representation for ASCII characters within an Ada program, and this representation is required to be the same for all implementations, it is host-independent in the sense that it will have the same meaning on any system. That does not imply that it can be successfully read into a VMM domain on any system. For example, a numeric value greater than SYSTEM.MAX_INT cannot be read successfully (although it can be "parsed" and ignored). This type of "incompatibility" is equally present in human-readable representations used for inter-host transport, as well.

Experience with a similar compressed representation used to transport separate compilation database structures from an MVS/370 system to an RSX-11/M system showed the compressed form to be approximately one quarter the size of linear graph notation (the human-readable form used on that system which is similar to the virtual record notation described here) with all superfluous blanks and null lines removed.

3.3 Adaptation

3.3.1 Initial Tool Construction

Development of a MAPSE with sufficient functionality and tools to support its own maintenance and further development requires several intermediate steps representing subsets of the end desired functionality. This section describes these steps, and the functionality of the various subsets. The end goal is a MAPSE that has sufficient functionality to support APSE development, MAPSE maintenance, and MAPSE tool maintenance independent of any other programming support environment. Specifically this includes bug fixes in both the KAPSE, the Ada Compiler, and all of the tools that make up the MAPSE. These functionalities exceed those necessary for embedded computer software development, but are necessary for a self-hosted MAPSE.

Since no MAPSE exists at present, primary development will take place on an IBM VM/370 system running the Conversational Monitoring System (CMS). The MAPSE itself runs under VM/370 with no additional operating system support other than that supplied by VM/370. The KAPSE is designed to run on a bare virtual machine. The set of issues and steps to bridge the gap between a CMS system and a self hosted MAPSE fall into a broad class of problems called "bootstrap" issues. MAPSE bootstrap issues fall into three distinct categories: (1) Compiler bootstrap - the Compiler must be written in Ada and hosted on the KAPSE; (2) KAPSE bootstrap - the KAPSE must be able to run on a bare virtual machine and support various MAPSE tools; (3) MAPSE tool bootstrap - certain of the MAPSE tools must be fully functional and run on CMS to support other bootstrap development. Separating the interdependencies of these three categories is necessary to allow otherwise independent development to occur in parallel. This separation is achieved by creating subsets of the MAPSE and KAPSE that run on CMS and provide sufficient functionality to allow parallel development. Portions of the code used to create these subsets will necessarily be discarded, but much larger portions will eventually form a part of the final system.

In the following sections, separate steps are described and a summary of the resultant functionality is provided. The bootstrap process allows coding and testing of Ada programs at a very early stage, both on CMS and on the bare virtual machine. This occurs long before the MAPSE Ada Compiler is fully functional, allowing KAPSE development to proceed in parallel with Compiler development. This is achieved by modifying the DARPA Ada Compiler to run on and generate code for the CMS operating system. The steps outlined below are presented in sequential order, representing larger and larger subsets of the full MAPSE functionality.

(a) Build an Ada Compilation Facility. The first step in creating a self-hosted MAPSE is to be able to write and debug programs written in Ada. Supplying this functionality on the development system (CMS) requires an Ada Compiler that runs on CMS and generates code that runs on CMS.

This is accomplished by modifying the DARPA Ada Compiler being built by Intermetrics, Inc. The DARPA Compiler is written in SIMULA and generates code for a DECSystem-10/20. This compiler will be rehosted to CMS, using the SIMULA compiler on the IBM system. The code generator is then modified to produce PL/I. This produces an Ada-to-PL/I Translator. The PL/I programs are then run through the PL/I compiler, linked, and executed in the CMS environment. This is known as the Bootstrap Compiler or initial compilation facility.

Functionality: This stage provides an Ada Translator hosted on and targeted for CMS with full Ada syntax and semantic analysis. All Ada language features except for tasking can be executed. This permits application program development, and unit testing of Ada packages that form various parts of the MAPSE. All portions of the MAPSE that are to be written in Ada can be started, including testing.

(b) Build a Mini-KAPSE and RTS on CMS. To support the development of the DataBase System and Ada Run Time System on CMS, a mini-KAPSE is built, simulating KAPSE functions on CMS. Higher level interfaces and implementation algorithms can be tested by using this primitive KAPSE simulation. These KAPSE functions are procedures that are loaded at run time with the PL/I run-time library, and eventually become KAPSE primitives.

Functionality: This stage permits the first set of integration tests and some functional tests to be written and tested for various parts of the MAPSE. Increased KAPSE-specific testing for all MAPSE components that depend closely upon the KAPSE can be started.

(c) Build Bare Virtual Machine Run Time System. To permit KAPSE modules to run on the bare virtual machine, a special run time system is built that completely replaces the PL/I run time system. This is an extension of the mini-KAPSE, and includes support of tasking and all Ada language features. The initial program load generation (IPLGEN) tool provides the capability to create a bare virtual machine load on the disks, a l allow a stand-alone Ada program to be loaded into the bare virtual machine using the VM/370 IPL command.

Functionality: This stage permits stand-alone KAPSE functional tests, as well as increased independence from CMS for MAPSE component testing. This Run Time System becomes part of the KAPSE and the Ada Program Run Time System.

(d) Incremental Development of VM/370. This stage includes the full development of the Ada Compiler written in Ada, the linker, the full bare virtual machine KAPSE, and the KAPSE version of the Ada Run Time System. Increasing functionality of one component facilitates further testing of its dependence on other MAPSE components. The MAPSE Linker is hosted on CMS and generates load modules for the KAPSE environment. A special tool (called Inject) inserts these load modules into the KAPSE Database so that the KAPSE can access them when it is running. The Compiler and Linker are designed so that host-specific interfaces are isolated; this permits ease of rehosting for these two MAPSE tools.

Functionality: When this stage is complete, CMS will be the host for cross compiling to the KAPSE environment. Full Ada will be supported, and the KAPSE will be complete.

(e) Rehost Ada Compiler, Link Editor and Other MAPSE Generation Tools. To complete the self-hosting of the MAPSE, the Ada Compiler, Linker, and their support tools must be hosted on the KAPSE. The Compiler and Linker are rehosted by modifing the separate portions that depend upon CMS. Specifically these include the VMM implementation package and file input/output operations. These tools will be cross compiled by the original Ada Compiler, and hosted on the KAPSE.

The other tools include LR, the tool used to generate the syntax analysis phase of the compiler; VMREP, the VMM representation analyzer; IPLGEN, the tool that generates an Initial Program Load module for the bare virtual machine; and ASM, the 370 machine code assembler. These tools will be re-written in Ada or ASM as appropriate and hosted on the MAPSE.

Functionality: When this stage is complete, the MAPSE is now complete, and is self hosted. It is capable of being regenerated without reference to any other programming support system. Self rehosting can be used as a functional test, as well as compiler validation.

3.3.2 Rehosting

The procedures for rehosting a MAPSE are demonstrated by moving the VM/370 MAPSE to a Perkin-Elmer 8/32 machine running OS/32. The fact that the MAPSE is completely self-supporting makes this job simpler than it might otherwise be, since all of the facilities of the MAPSE are available, and the elements that must be moved are already written in Ada.

The basic procedure is to develop a cross-compilation, linking, and loading facility, treating the 8/32 as the target machine. This is essentially the same as step (d) above, except that the "development" consists of retargetting the compiler's code generator, modifying the Linker to produce an output that can be loaded on the 8/32, and re-writing those portions of the run-time system that depend on VM/370 so that they can run under OS/32. Once this has been done, programs can be compiled and linked on VM/370 and run on the 8/32 system.

The final step is similar to step (e), above, except that modifying the VMM implementation package and input-output package appropriately for the 8/32 environment and cross-compiling now can be used to move nearly the entire MAPSE. (A small number of additional parameters, e.g., number of buffers used by tools, will also need to be modified in each tool.) The only addition needed to make this MAPSE completely self-supporting would be to make the OS/32 assembler invokable from the MAPSE.

PAGE LEFT BLANK INTENTIONALLY

4.0 QUALITY ASSURANCE PROVISIONS

A quality product is assured by a combination of sound design, careful implementation, and effective testing. The CPDP provides a detailed discussion of the methodologies employed to achieve this combination of goals in the development of the MAPSE. This section identifies some specific issues of quality that are particularly significant for the text parsing and data management facilities of the MAPSE and defines unit, integration, and acceptance test levels for them. Before considering them individually, it is worthwhile to note that the recognition of these facilities as design elements in itself contributes to the overall quality of the MAPSE.

4.1 Text Parsing

The lexer and parser generators (LEX and LR) are instance of tools commonly used in modern compiler construction. Their design and implementation are largely borrowed from existing tools that are demonstrably "correct". The primary quality metrics beyond correctness are then the error recovery characteristics of the generated parser and the overall level of helpfulness the tools provide in terms of grammar debugging and documentation aids. The tools chosen for implementation in the MAPSE exhibit both superior error recovery in the generated parser and proven aids to grammar debugging and documentation.

4.1.1 Unit Testing

LEX and LR are unit tested by processing sets of simple lexical specifications and grammar and examining the generated automata. Both positive (error-free input) and negative (erroneous input) tests are performed. Unit tests can be performed using the initial compilation facility under CMS (see Section 3.3).

4.1.2 Integration Testing

Integration testing consists of generating a complete parser and lexer for the Ada grammar, combining the sources, and compiling the result using the initial compilation facility. The resulting program is then run under CMS with reduction tracing enabled. Successful testing at this level permits a crossover in the development and testing of the compiler's LEXSYN phase, eliminating interim tools used to produce the lexer and parser. The environment in which the generator programs run is not significant, and could remain the CMS environment until acceptance testing. However, the generated lexer and parser must run in each of the intermediate environment defined for the compiler.

4.1.3 Acceptance Testing

The acceptance test for LEX and LR consists of recompiling the generator programs using the MAPSE-hosted compiler, regenerating the lexer and parser using the new generators, and recompiling the LEXSYN phase of the compiler. Acceptance of the compiler created by linking with the new version of LEXSYN defines acceptance of LEX and LR.

4.2 Data Management

The VMM implementation supports most other MAPSE tools, providing access to their data. Besides freedom from error, significant quality metrics for VMM include "robustness" and error detection capabilities.

Robustness is used here to indicate a certain resilience in handling errors; in particular, it applies to errors at the level of the human interface. This level is manifested in the representation analyzer, which must read virtual record descriptions prepared by people, and in the conversion operations which read and write the human-readable virtual record notation.

Both the representation analyzer and the operations that read virtual record notation must provide useful error diagnostics and recovery mechanisms that permit processing to continue until the complete input is processed. In the case of the operations that write virtual record notation, the presence of inconsistencies or errors in the internal representation should be noted, but should not inhibit output since the primary use of virtual record notation is as a debugging aid.

Detection and identification of errors here refers primarily to incorrect or inconsistent use of VMM operations by a tool. Many of these kinds of errors are detected at compile time due to Ada's strong typing and the use of derived types to specify the classes of objects designated by locators. Other errors can be detected by the VMM implementation, such as dereferencing null locators type, out-of-bounds indexing of VMM arrays, etc. These kinds of errors would almost inevitably result in errors being detected by Ada constraint-checking at some later point in program execution; however, it is important that the VMM implementation detect these errors at the place that they are first introduced.

4.2.1 Unit Testing

Unit testing is performed on the VMM implmentation package. The package is first tested by driver modules that exercise its primitive "typeless" allocation and dereference operations, progressing from the single domain single sub-domain case to multiple domains and sub-domains. Subsequent unit tests are performed for each directly-supported data abstraction to verify its abstract properties. These tests use hand-coded interface modules for those operations that are generated by the representation analyzer in later stages of development.

4.2.2 Integration Testing

Integration testing is performed using an initial representation analyzer constructed by modifying the bootstrap compiler implemented in Simula (see Section 3.3). The virtual record type definition for the VREC_DESCRIPTION virtual record type is input to this version of the representation analyzer, and the access package produced is used to test the facilities for writing virtual record notation. The test consists of adding a procedure to the access package that calls the build_VREC_DESCRIPTION_description procedure and then calls the vrn_output procedure, passing it the domain value used by the access package to reference the description (see Section 3.2.2.3). The VRN output produced when the package is compiled and the test procedure executed is then examined.

Once this test is completed, the virtual record type definition for the Diana implementation is processed, and the generated access package is used to begin construction and testing of compiler phases. Once the LEXSYN and SEM phases are completed, the representation analyzer can be written in Ada using those phases and the access package for Diana. When the Ada version of the representation analyzer is complete, the integration test is repeated for the new version, and the initial version may be discarded.

4.2.3 <u>Acceptance Testing</u>

Acceptance testing for VMM is largely composed of acceptance testing for the MAPSE tools which use it. However, a series of formal tests for each of the supported data abstractions, the conversion operations for virtual record notation and compressed binary form, and for specific domain and sub-domain operations will be defined.

5.0 PREPARATION FOR DELIVERY (N/A)

6.0 NOTES

None.

10.0 APPENDIX I

10.1 Virtual Record Notation Grammar

The following grammar for virtual record notation is given in a simple variant of BNF, as follows:

1. optional items are enclosed in square brackets;

2. curly braces enclose items that may be repeated zero or more times;

3. double quotes enclose characters that are not to be considered BNF meta-characters (e.g. "<").

The lexical structure of terminals is basically the same as that defined by Ada, where applicable: <integer>, <float>, <string>, <identifier>. The compound symbols (., .), (*, and *) are constructed from the Ada basic character set and may be replaced by the single characters [,], {, and } when these characteristics are available. The commenting convention differs from Ada, allowing comments to be embedded within lines, and to span lines: comments are both introduced and terminated by a vertical bar (or exclamation point, depending on the available character set). This convention allows programs which generate virtual record notation to place comments beside a token without altering the line structure, and it allows a human reader to "comment out" portions of lines easily when using VRN for debugging and testing purposes.

```
<domain>             ::= {<sub-domain>};

<sub-domain>         ::= <declare part> <record part> END

<declare part>       ::= DECLARE <use reference> <use decl>{<use decl>}

<use decl>           ::= <label> / USE (. <components> .)

<record part>        ::= RECORD <record reference> {<record def>}

<use reference>      ::= <label> "<-"

<record reference> ::= <label> / <label> "<-" | <label> "<-"

<label>              ::= <integer> | <identifier>

<record def>         ::= <label>: <identifier> <record>

                         | <label>: <aggregate>

<record>             ::= (. [<components>] .)

<aggregate>          ::= <array> | <list> | <set> | <string>

<components>         ::= <component> {;<component>}

<component>          ::= <identifier> "=>" <value>

<value>              ::= <record> | <basic value>

<basic value>        ::= TRUE | FALSE | <integer> | <float> | <string>
                         | <identifier> | <record reference>
                         | <array> | <list> | <set>

<array>              ::= ( [<sequence>] )

<list>               ::= "<" [<sequence>] ">"

<set>                ::= (* [<sequence>] *)

<sequence>           ::= [<size>] {<basic value>}

<size>               ::= :<integer>
```